

Support de cours

Cours:

Introduction à la programmation orientée objet (en C++)

Vidéo:

W14-04-copieprofonde-CPP-pt2

Concepts (extraits des sous-titres générés automatiquement) :

Copie membre. Construction d'un objet. Emplacements mémoire nécessaires. Constructeur de la classe. Rectangle r. Copie de surface. Hauteur du rectangle. Zones mémoire. Emplacement mémoire. Objet de type. Terme de l'exécution de cette ligne. Niveau du destructeur de la classe. Rectangle tmp. Définition explicite du constructeur. Définition de la classe.



[vers la recherche de séquences vidéo](#)

(dans Introduction à la programmation orientée objet (en C++).)



[vers la vidéo](#)

Center for Digital Education. Plus de matériel de soutien pédagogique ici :

<https://www.epfl.ch/education/educational-initiatives/cede/educational-technologies-gallery/boocs-en/>

Copie profonde

(Partie 2)

Introduction à la programmation orientée objet (en C++)

Jean-Cédric Chappelier, Jamila Sam et Vincent Lepetit

...

notes

résumé

0m 0s



Petit rappel (2)

Dans certains cas, les versions minimales par défaut des méthodes constructeurs/destructeurs **ne sont pas adaptées** : exemple du *comptage des instances* (cf. semaine passée).

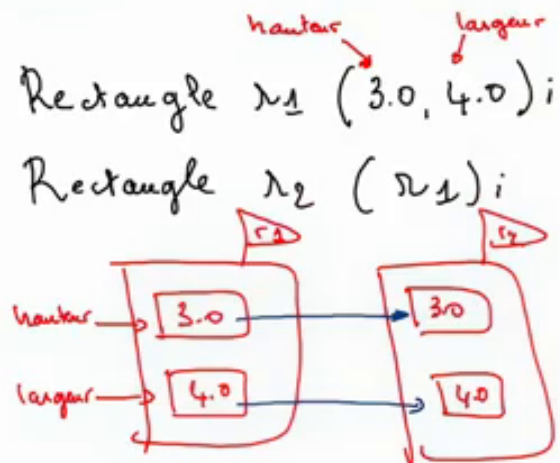
Autre exemple :

Le *constructeur de copie par défaut* réalise une copie membre à membre des attributs

→ **copie de surface**

Ceci pose typiquement problème lorsque **certain attributs** de la classe sont des **pointeurs**.

Examinons pourquoi sur un exemple concret...



Lorsque les attributs sont de type « pointeur »,

notes

résumé

0m 1s



Soit une autre définition possible (farfelue, mais possible !) de classe `Rectangle` :

```
class Rectangle {  
private:  
    double* largeur; // aïe, un pointeur !  
    double* hauteur;  
public:  
    Rectangle(double l, double h)  
        : largeur(new double(l)), hauteur(new double(h)) {}  
    ~Rectangle() { delete largeur; delete hauteur; }  
    double getLargeur() const;  
    double getHauteur() const;  
    // ...  
};
```

contrairement au cas que nous avons ici sous les yeux, et bien cette copie membre à membre, cette copie de surface, peut poser un certain nombre de problèmes, c'est ce que nous allons étudier dans ce qui suit. Prenons un exemple concret,

notes

résumé

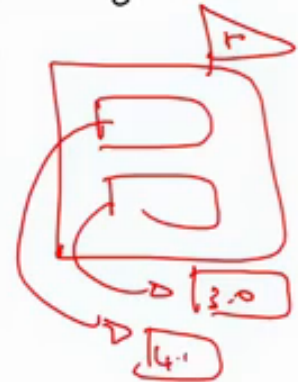
0m 5s



Soit une autre définition possible (farfelue, mais possible !) de classe `Rectangle` :

```
class Rectangle {
private:
    double* largeur; // aïe, un pointeur !
    double* hauteur;
public:
    Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
    ~Rectangle() { delete largeur; delete hauteur; }
    double getLargeur() const;
    double getHauteur() const;
    // ...
};
```

`Rectangle r(3.0, 4.0);`



celui d'une classe « `Rectangle` » où l'on aurait choisi d'implémenter les attributs « `largeur` » et « `hauteur` » au moyen de pointeurs sur des « `double` » au lieu de « `double` » tout simplement ; il s'agit bien sûr d'une implémentation qui n'est pas très naturelle ni très justifiée mais nous la prenons ici uniquement pour illustrer notre propos sur un exemple simple. Le constructeur de la classe « `Rectangle` » est en charge d'allouer dynamiquement les emplacements mémoire nécessaires à stocker la largeur et la hauteur d'une instance de « `Rectangle` ». Pour rappel, cette tournure permet d'allouer dynamiquement, pendant que le programme s'exécute, un emplacement mémoire pour un « `double` », d'y stocker la valeur « `l` », ce qui est retourné par cette instruction et l'adresse du « `double` » en question, laquelle est ici stockée dans l'attribut « `largeur` » qui est effectivement de type pointeur donc une adresse. En utilisant cette définition de la classe « `Rectangle` », il est donc possible, dans une autre portion de programme, de déclarer « initialiser un rectangle », en utilisant le constructeur fourni, comme ceci, le résultat d'une telle déclaration « initialisation » correspondra à la construction d'un objet en mémoire dont les deux champs « `largeur` » et « `hauteur` » sont des adresses vers des « `double` »

notes

résumé

0m 19s



Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

représentant la largeur et la hauteur du rectangle. En C++, celui qui alloue a la responsabilité de désallouer, ici la classe « Rectangle » prend la responsabilité d'allouer des emplacements mémoire au moment où l'on crée un « Rectangle » elle doit donc, lorsque cet objet cesse d'exister, désallouer les emplacements nécessaires et ceci se fait bien évidemment au niveau du destructeur de la classe « Rectangle ». Vous noterez enfin que dans cette définition de la classe « Rectangle » il y a une définition explicite du constructeur, également une définition explicite du destructeur mais aucune définition explicite du constructeur de copie ce qui signifie que c'est la version par défaut qui sera employée le cas échéant.

notes

résumé

1m 37s



Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

Rectangle r(3.0, 4.0)

Supposons maintenant qu'un programmeur souhaite utiliser la classe « Rectangle » et donc il définit une fonction « afficher_largeur » qui prend en paramètre un « Rectangle » et qui a pour but d'en afficher la largeur ; donc typiquement il pourrait décider, dans une portion de son programme, de déclarer un objet de type « Rectangle » qui serait construit au moyen du constructeur

notes

résumé

2m 20s



Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

*Rectangle r(3.0, 4.0);
afficher_largeur(r);*

que l'on a présenté dans le transparent précédent et puis d'invoquer la fonction « afficher_largeur », sur ce « Rectangle », pour en afficher la largeur.

notes

résumé

2m 45s

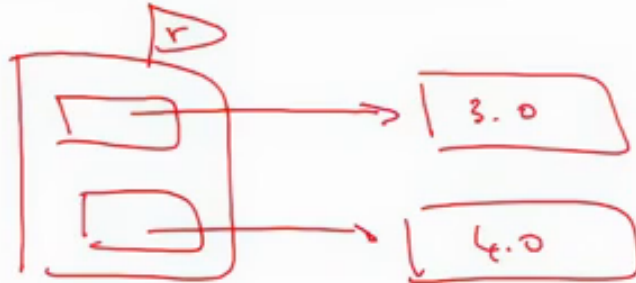


Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

Rectangle r(3.0, 4.0);
afficher_largeur(r);



En mémoire, nous venons de le voir, l'objet « r » a l'allure suivante donc ses champs « largeur » et « hauteur » pointent vers, ce sont des adresses qui pointent vers deux emplacements mémoire, contenant les valeurs de la largeur et de la hauteur. C'est ce qui est construit au terme de l'exécution de cette ligne,

notes

résumé

2m 55s

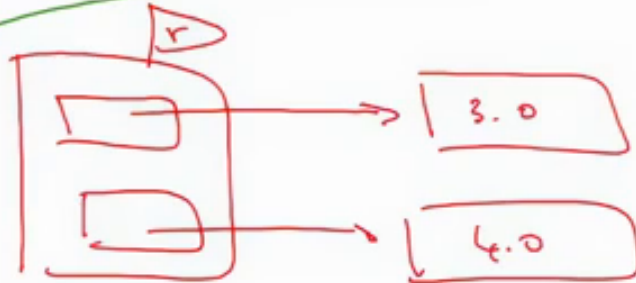


Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

→ Rectangle r(3.0, 4.0);
→ afficher_largeur(r);



on souhaite ensuite faire afficher la largeur du rectangle ainsi construit et en fait ici on risque d'avoir de mauvaises surprises, on risque de se rendre compte que l'objet « r » n'est pas sorti indemne de cet affichage mais que s'est-il donc passé ici lors de l'exécution de la fonction « afficher_largeur » qui se borne pourtant à afficher la largeur de notre « Rectangle ». Si l'on examine l'entête de la fonction « afficher_largeur », on voit que le paramètre, le « Rectangle » dont on veut faire afficher la valeur est en fait passé « par valeur ». Ceci signifie concrètement que lorsque l'on exécute cette ligne de code la valeur de « r » va être copiée dans « tmp » puisqu'on a à faire à un passage par valeur

notes

résumé

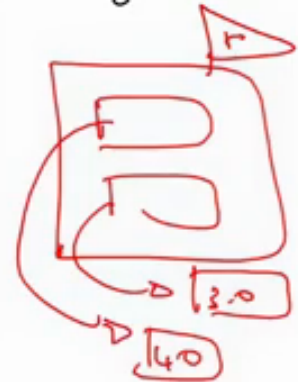
3m 14s



Soit une autre définition possible (farfelue, mais possible !) de classe `Rectangle` :

```
class Rectangle {  
private:  
    double* largeur; // aïe, un pointeur !  
    double* hauteur;  
public:  
    Rectangle(double l, double h)  
        : largeur(new double(l)), hauteur(new double(h)) {}  
    ~Rectangle() { delete largeur; delete hauteur; }  
    double getLargeur() const;  
    double getHauteur() const;  
    // ...  
};
```

`Rectangle r(3.0, 4.0);`



et le code de la fonction « affiche » s'exécute sur la copie « tmp ». Qui est en charge de réaliser cette copie du « Rectangle r » dans le « Rectangle tmp » ?

notes

résumé

3m 59s



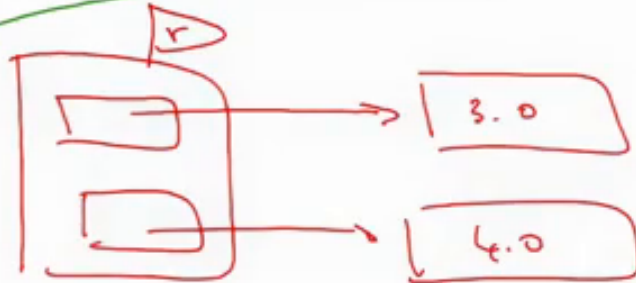
Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

copie ?

→ Rectangle r(3.0, 4.0);
→ afficher_largeur(r);



Et bien, oui c'est bien le constructeur de copie de la classe « Rectangle » qui est en charge de réaliser cette copie. Celui-ci n'est pas défini explicitement, on vous le rappelle, ce qui veut dire que c'est le constructeur de copie par défaut, qui est invoqué lequel fait une copie de la surface,

notes

résumé

4m 14s



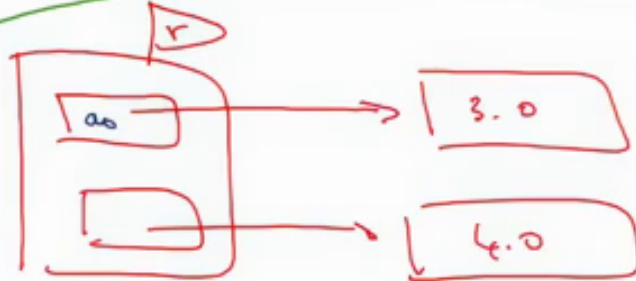
Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

copie ?

→ Rectangle r(3.0, 4.0);
→ afficher_largeur(r);



donc au moment de l'exécution de cette ligne de code il va y avoir création d'un objet « tmp » créé par copie à partir de l'objet « r » au moyen du constructeur de copie par défaut. Ceci signifie que l'objet « tmp » aura cette allure le constructeur par défaut fait une copie membre à membre

notes

résumé

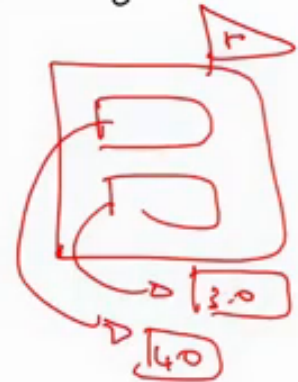
4m 28s



Soit une autre définition possible (farfelue, mais possible !) de classe `Rectangle` :

```
class Rectangle {
private:
    double* largeur; // aïe, un pointeur !
    double* hauteur;
public:
    Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
    ~Rectangle() { delete largeur; delete hauteur; }
    double getLargeur() const;
    double getHauteur() const;
    // ...
};
```

`Rectangle r(3.0, 4.0);`



des valeurs des attributs donc supposons que le premier attribut contienne la valeur « adresse 1 » et le second « adresse 2 », il y aura une copie membre à membre ce qui signifie concrètement que les deux champs de l'objet « tmp » pointent vers les mêmes zones mémoire que ceux de l'objet « r ». Le corps de la fonction « `afficher_largeur` » peut s'exécuter sans problème. A partir de l'objet « tmp », on peut accéder via le pointeur à la valeur du champ « largeur » qui est la même que celui du rectangle et dans ce cas-là l'affichage va correspondre à ce que l'on souhaite. C'est un petit peu plus loin que les choses vont commencer à se gâter. En effet, le paramètre « tmp » de la fonction « `afficher_largeur` » est un objet qui est local à la fonction et qui ne peut pas être utilisé à l'extérieur de la fonction, ceci signifie qu'au moment où l'on termine l'exécution de la fonction « `afficher_largeur` » cet objet « tmp » n'a plus de raison d'exister et donc le destructeur va être invoqué automatiquement sur cet objet « tmp ».

notes

résumé

4m 49s



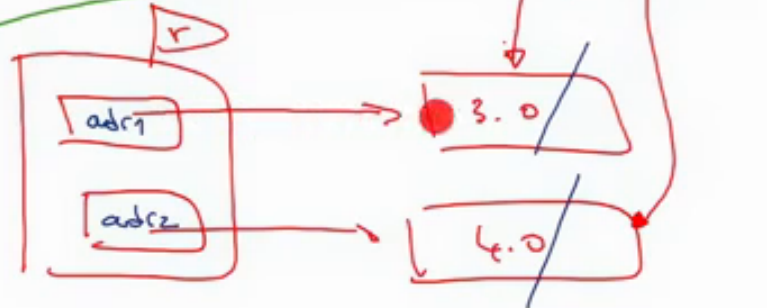
Exemple

Que se passe-t-il lorsqu'on invoque la fonction suivante ?

```
void afficher_largeur(Rectangle tmp)
{
    cout << "Largeur: " << tmp.getLargeur() << endl;
}
```

copie ?

→ Rectangle r(3.0, 4.0);
→ afficher_largeur(r);



Pour rappel, le destructeur de la classe « Rectangle » a pour vocation de désallouer les zones mémoire associées aux attributs de l'objet. Dans le cadre de notre exemple ceci signifie qu'au moment où la fonction « afficher_largeur » a terminé son exécution, le destructeur est invoqué sur l'objet « tmp », lequel va désallouer les emplacements mémoire associés à ses attributs. Or, ici, il se trouve que malheureusement les zones mémoire désallouées lors de la destruction de « tmp » sont les mêmes que

notes

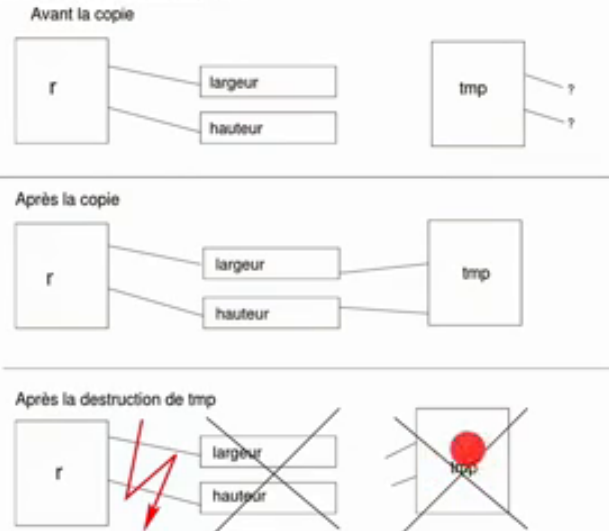
résumé

5m 52s



Exemple (3)

Voilà ce qui se produit concrètement :



il faut **redéfinir le constructeur de copie** de sorte à ce qu'il **duplique véritablement les champs concernés** → **copie profonde**

celles qui sont pointées par notre objet « Rectangle r » qui, lui, continue bel et bien son existence une fois que la fonction « `afficher_largeur` » a terminé son exécution. En réalité, il n'y a pas de raison que « `afficher la largeur` » d'un « Rectangle » cause la destruction des zones mémoire associées à ce « Rectangle » c'est pourtant bel et bien ce qui s'est passé ici et ceci veut dire très concrètement que si l'on essaie d'utiliser le « Rectangle » ici pour faire une autre manipulation dessus, ce qui normalement en principe ne devrait pas être empêché par l'affichage de sa largeur précédemment et bien on est en train ici de travailler avec un objet « Rectangle r » qui est corrompu puisque les zones mémoire associées ont été désallouées. Pour résumer notre exemple, appeler la fonction « `afficher_largeur` » en lui passant en argument un « Rectangle r » fait que « r » est copié dans « tmp » en raison du passage par valeur, lorsque la fonction « `afficher_largeur` » a terminé son exécution, « tmp » qui est local à la fonction, doit être détruit, il l'est automatiquement par le destructeur de la classe « Rectangle » ce destructeur va libérer la mémoire pointée par les champs « `largeur` » et « `hauteur` » de « tmp » qui se trouvent être les mêmes zones mémoire pointées par les champs « `largeur` » et « `hauteur` » de « r ». Du coup, toute utilisation de « r » après que la fonction « `afficher_largeur` » ait terminé son exécution nous expose à un certain nombre de dangers, on peut voir le contenu de « r » changer de façon intempestive, on s'expose également à un risque de « Segmentation Fault » s'il y a des opérations

notes

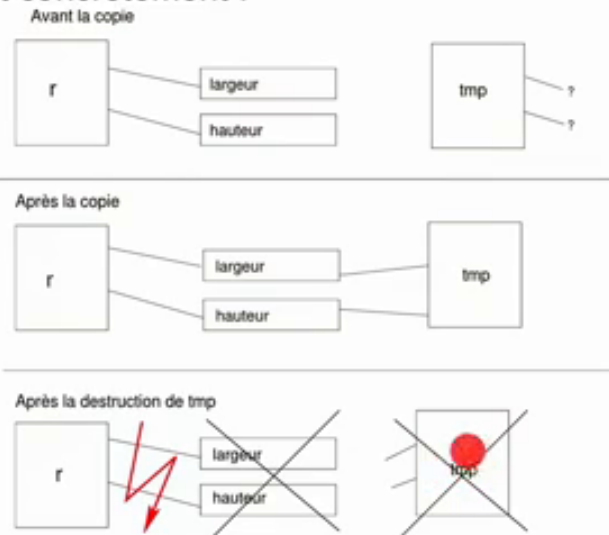
résumé

6m 25s



Exemple (3)

Voilà ce qui se produit concrètement :



il faut **redéfinir le constructeur de copie** de sorte à ce qu'il **duplique** véritablement les champs concernés → **copie profonde**

de destruction sur cet objet « r » qui interviennent par la suite. La corruption de l'objet « r »,

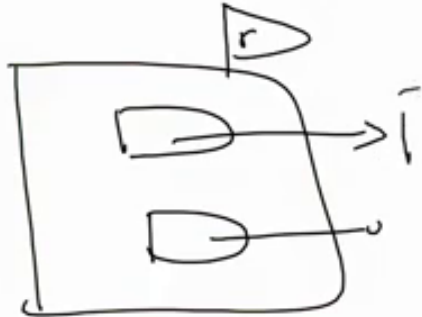
notes

résumé

Exemple (4)

Une bonne solution consiste alors à **redéfinir le constructeur de copie** :

```
Rectangle(const Rectangle& obj)
: largeur(new double(*(obj.largeur))) ,
  hauteur(new double(*(obj.hauteur)))
{}
```



suite à la destruction de l'objet « tmp », est en fait liée simplement au fait que ces deux objets pointent sur les mêmes zones mémoire alors qu'ils ne le devraient pas ; « r » et « tmp » pointent vers les mêmes zones mémoire car comme nous l'avons vu précédemment la copie qui est réalisée est une copie de surface. Pour éviter le problème il faudrait que la copie fasse ce que l'on appelle une copie profonde c'est-à-dire qu'elle ne se contente pas de copier les adresses mais qu'elle copie plutôt les contenus pointés ; donc concrètement ceci voudrait dire que l'on ne veut pas avoir de lien entre « tmp » et les zones mémoire pointées par « r » mais l'on veut créer à partir de « r » un objet « tmp » qui pointerait sur deux zones mémoire indépendantes qui aurait simplement les mêmes valeurs que celles de l'objet d'origine, à savoir « r ». Du coup, la destruction du « tmp », après la fin de l'exécution de la fonction « afficher_largeur », n'aurait d'incidence que sur les champs pointés par « tmp » et n'aurait aucune incidence sur ceux pointés par l'objet « r » lequel pourrait continuer à être utilisé sans problème par la suite. Le constructeur de copie par défaut qui fait une copie de surface n'est donc pas adapté ; il s'agit donc ici de redéfinir le constructeur de copie de sorte à ce qu'il fasse une copie profonde qui duplique des zones pointées. En souhaitant que le constructeur de copie duplique les zones pointées,

notes

résumé

8m 1s



Exemple (4)

Une bonne solution consiste alors à **redéfinir le constructeur de copie** :

```
Rectangle(const Rectangle& obj)
: largeur(new double(*(obj.largeur))) ,
  hauteur(new double(*(obj.hauteur)))
{}
```



donc à partir d'un « Rectangle r » dont les attributs seraient des pointeurs vers des « double » particuliers. On souhaite que l'objet « tmp » obtenu par copie depuis « r » soit un objet complètement indépendant dont la valeur des attributs serait de nouvelles adresses pointant vers des « double » dont la seule particularité serait d'avoir des valeurs identiques à celles de l'objet d'origine. L'objet copié étant ainsi complètement indépendant de l'objet d'origine, sa destruction n'a plus aucune incidence sur l'objet d'origine. Le constructeur de copie doit donc être réécrit de sorte à dupliquer les zones pointées. Créer une copie « tmp » de l'objet « r » en utilisant ce constructeur de copie se traduirait donc par la situation suivante en mémoire l'objet « tmp » aurait un champ « largeur » dont la valeur serait l'adresse d'un objet nouvellement alloué donc une nouvelle adresse et dont le contenu pointé

notes

résumé

9m 37s

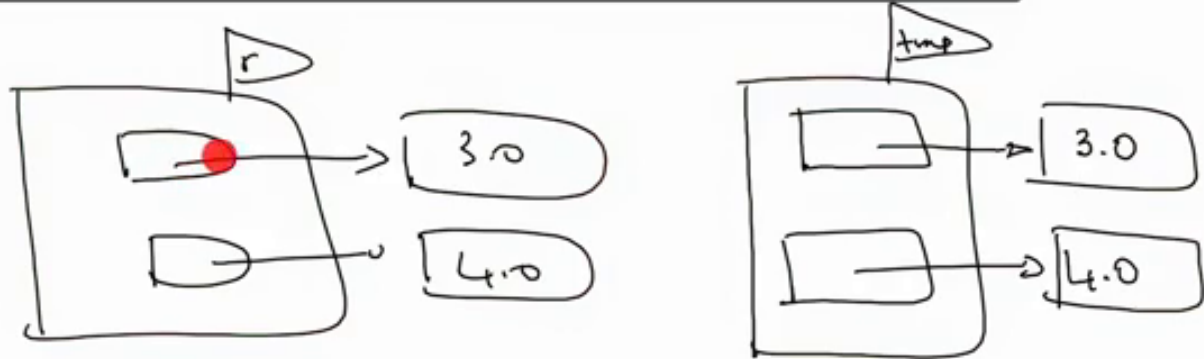


Exemple (4)

Une bonne solution consiste alors à **redéfinir le constructeur de copie** :

```
Rectangle(const Rectangle& obj)
: largeur(new double(*(obj.largeur))) ,
  hauteur(new double(*(obj.hauteur)))
{}

```



serait obtenu par copie du contenu pointé par l'objet d'origine et pareil pour la triple hauteur. Donc ici si l'on veut décortiquer un peu les choses « obj » est l'objet copié, l'objet à partir duquel on veut faire la copie, cette notation « obj.largeur » va nous donner la valeur de cette adresse,

notes

résumé

10m 49s



Exemple : Définition complète de la classe

```
class Rectangle {
public:
    Rectangle(double l, double h)
        : largeur(new double(l)), hauteur(new double(h)) {}
    Rectangle(const Rectangle& obj);
    ~Rectangle();
    // Note: il faudrait aussi redefinir operator= !
private:
    double* largeur;      double* hauteur;
};
// constructeur de copie
Rectangle::Rectangle(const Rectangle& obj)
    : largeur(new double(*(obj.largeur))),
      hauteur(new double(*(obj.hauteur)))
{}
// destructeur
void Rectangle::~~Rectangle() {
    delete largeur;
    delete hauteur;
}
```

« *obj.largeur » va nous donner le contenu pointé par cette adresse ; cette valeur correspond donc au contenu pointé par l'objet d'origine, l'instruction « new double » ici va donc allouer un emplacement mémoire pour le champ « largeur » avec pour contenu le contenu de l'objet d'origine.

notes

résumé

11m 13s



Pour conclure

- ▶ Si une classe contient des pointeurs, penser à la copie profonde (**au moins** se poser la question) :

- ▶ constructeur de copie ;
- ▶ surcharge de l'opérateur = ;
- ▶ destructeur.



- ▶ Remarque : si l'on redéfinit le constructeur de copie d'une sous-classe, penser à explicitement mettre l'appel au constructeur **de copie** de la super-classe (sinon c'est le constructeur **par défaut** de la super-classe qui est appelé !!)

Exemple :

```
Rectangle(const Rectangle& obj)
: FigureGeometrique(obj), // ...etc.
{ // ...etc.
}
```

Donc voici comment il faudrait réécrire la classe « Rectangle » de sorte à s'éviter les problèmes de copie de surface lorsque les attributs sont des pointeurs nous avons vu que le fait d'avoir un constructeur qui s'occupe d'allouer la mémoire et un destructeur qui s'occupe de la restituer n'a pas suffi, il faut en plus définir un constructeur de copie qui fait de la copie profonde c'est-à-dire qui ne copie pas les adresses mais qui duplique les zones pointées. En toute rigueur il faudrait aussi redéfinir « opérateur = » qui comme nous l'avons vu dans les séquences précédentes est très proche du constructeur de copie. La version par défaut de l'opérateur « = » fait elle aussi en effet, une copie de surface et si on ne la définit pas proprement aux endroits voulus et bien on s'expose aux mêmes problèmes qu'en s'abstenant de redéfinir correctement le constructeur de copie. Voilà, du point de vue de la méthodologie lorsque une classe contient des attributs qui sont des pointeurs il faut penser à la copie profonde, au moins se poser la question : est-ce qu'on doit redéfinir le constructeur de copie de sorte à ce qu'il fasse de la duplication des zones mémoire pointées, ou non ; si l'on redéfinit le constructeur de copie il faut penser également à la surcharge de l'opérateur « = » qui doit être défini de façon appropriée et lorsque l'on fait de la copie profonde on est souvent amené à utiliser de nouvelles ressources en mémoire et dans ce cas il faut penser également à adapter le destructeur. à adapter le destructeur.

notes

résumé

11m 38s

